The CEL Manual (v0.9b)

Boontawee Suntisrivaraporn

Institute for Theoretical Computer Science
TU Dresden, Germany
meng@tcs.inf.tu-dresden.de

July 10, 2006

Abstract

Description logics (DLs) are an important family of formalisms for reasoning about ontologies. CEL[1] is a reasoner for the description logic $\mathcal{EL}^+$, supporting as its main reasoning task the computation of the subsumption hierarchy induced by $\mathcal{EL}^+$ ontologies. The most distinguishing feature of CEL is that, unlike other modern DL reasoners, it implements a tractable (i.e., polynomial-time) algorithm. The supported description logic $\mathcal{EL}^+$ offers a selected set of expressive means that are tailored towards the formulation of medical and biological ontologies.

Contents

1 Introduction to CEL **1**

2 An Overview of CEL Ontologies **3**

3 Macros and Functions **5**
3.1 Concept and Role Construction 5
3.2 Knowledge Base Declarations 6
3.3 Operations on the Ontology (TBox) 9
3.4 Query Commands 10
3.5 Other Commands 13

4 An Example: Learning by Doing **15**

1 Introduction to CEL

The description logic $\mathcal{EL}^{++}$ introduced in [1] is tailored towards the formulation of medical and biological ontologies. A distinguishing feature of $\mathcal{EL}^{++}$ compared to other

[1] which stands for a polynomial-time Classifier for the description logic $\mathcal{EL}^+$

description logics such as $\mathcal{SHIQ}$ and OWL [3, 4] is that, despite offering considerable expressivity, reasoning in $\mathcal{EL}^{++}$ is tractable, i.e., can be performed in polynomial time. This is clearly an advantage over the ExpTime worst-case complexity of reasoning in $\mathcal{SHIQ}$ and OWL.

The ultimate goal of CEL is to provide a highly efficient reasoner for the description logic $\mathcal{EL}^{++}$. A main emphasis of CEL is on reasoning with ontologies formulated in $\mathcal{EL}^{++}$, in particular on computing the subsumption hierarchy induced by such an ontology. As of now, CEL implements a practically useful subset of $\mathcal{EL}^{++}$ that we call $\mathcal{EL}^{+}$. More precisely, $\mathcal{EL}^{+}$ provides for the following concept constructors:

- top concept "$\top$";
- bottom concept "$\bot$", which boils down to the expressivity of concept disjoinhtness axioms (see below);
- conjunction "$C \sqcap D$";
- existential restriction "$\exists r.C$".

For building up ontologies, the following means of expressivity are available:

- primitive concept definitions "$A \sqsubseteq D$";
- concept definitions "$A \equiv D$";
- general concept inclusion (GCI) axioms "$C \sqsubseteq D$";
- concept equivalence axioms "$C \equiv D$";
- concept disjointness axioms "$C_1 \sqcap \cdots \sqcap C_n \sqsubseteq \bot$";
- role domain restrictions "$\exists r.\top \sqsubseteq C$";
- transitivity assertions for roles "$r \circ r \sqsubseteq r$";
- role hierarchy axioms "$r \sqsubseteq s$";
- right-identity rules "$r \circ s \sqsubseteq r$" and left-identity rules "$r \circ s \sqsubseteq s$";
- role inclusion (RI) axioms of the form "$r \circ s \sqsubseteq t$".

The ontology classification algorithm implemented in CEL has been proposed in [2]. This algorithm is a refinement for implementation purposes of the well-known polynomial-time algorithm for computing classification in $\mathcal{EL}^{++}$ giving in [1]. Currently, CEL accepts inputs in a slight extension of the KRSS (Knowledge Representation System Specification) syntax [5]. The following system requirements are assumed:

- Linux operating system;
- Physical memory at least 128MB.[2]

[2]Considerably more memory may be needed for larger ontologies.

CEL is available as a precompiled binary package which can be run on Linux platforms. The package can be obtained from:

`http://lat.inf.tu-dresden.de/systems/cel/`

The package consists of the CEL executable, this user manual, the related papers [1, 2], and some example $\mathcal{EL}^+$ ontologies.

This manual details how to get CEL up and running, how to create an ontology, and how to classify it. In Section 2, an overview of the concept and ontology language of CEL is given. In Section 3, all commands offered by CEL will be described in detail, including syntax and helpful examples. Finally, Section 4 gives a step-by-step introduction to CEL at work.

2 An Overview of CEL Ontologies

Concepts and roles are built up from sets of concept and role names (which can be freely chosen by the user), using the concept and role constructors provided by $\mathcal{EL}^+$. The following syntax rules describe the formation of concepts and roles in CEL:

$$C \longrightarrow CN \mid \mathsf{top} \mid \mathsf{bottom} \mid (\mathsf{and}\ C_1 \ldots C_n) \mid (\mathsf{some}\ RN\ C)$$

$$R \longrightarrow RN \mid (\mathsf{compose}\ RN_1\ RN_2)$$

where CN, C, RN, and R (all possibly with subscripts) range over concept names, concepts, role names, and roles, respectively.

A more detailed description of the syntax along with helpful examples is given in Section 3.1.

We now give an overview of the axioms that are available for building up ontologies. Such axioms can be divided into concept axioms and role axioms.

Primitive concept definitions state the necessary condition of a concept name.
DL notation: $CN \sqsubseteq C$
CEL syntax: (define-primitive-concept CN C)

Concept definitions state the definition (both necessary and sufficient conditions) of a concept name.
DL notation: $CN \equiv C$
CEL syntax: (define-concept CN C)

General concept inclusions state the subsumption (or containment) between two concept descriptions.
DL notation: $C_1 \sqsubseteq C_2$
CEL syntax: (implies C_1 C_2)

Concept equivalence axioms state the equivalence between two concept descriptions.
DL notation: $C_1 \equiv C_2$
CEL syntax: (equivalence C_1 C_2)

Concept disjointness axioms assert the disjointness of two or more concept descriptions.
DL notation: $C_1 \sqcap \cdots \sqcap C_n \sqsubseteq \bot$
CEL syntax: (disjoint C_1 $\cdots$ C_2)

Role domain restrictions assert the domain of a role name
DL notation: $\exists RN.\top \sqsubseteq C$
CEL syntax: (define-primitive-role RN :domain C)

Role transitivity axioms assert the transitivity of a role name
DL notation: $RN \circ RN \sqsubseteq RN$
CEL syntax: (define-primitive-role RN :transitive t)

Role hierarchies assert the relationship between a role name and its super role name.
DL notation: $RN_1 \sqsubseteq RN_2$
CEL syntax: (define-primitive-role RN_1 :parent RN_2)

Right-identity (left-identity) rules specifies a right (left) identity of another role.
DL notation: $RN_1 \circ RN_2 \sqsubseteq RN_1$
CEL syntax: (define-primitive-role RN_1 :right-identity RN_2); and,
DL notation: $RN_1 \circ RN_2 \sqsubseteq RN_2$
CEL syntax: (define-primitive-role RN_2 :left-identity RN_1)

Role inclusions state the subsumption (or containment) between a role composition on the lhs and a role name on the rhs.
DL notation: $RN_1 \circ RN_2 \sqsubseteq RN_3$
CEL syntax: (role-inclusion (compose RN_1 RN_2) RN_3)

We note that only `role-inclusion` and `implies` are sufficient to express any $\mathcal{EL}^+$ ontologies. The additional commands are included for ease of use and for compatibility with other DL reasoners.

From version `0.9beta` on, CEL comes with simple command line options, where the user can classify his/her ontology and then output the classification results in the format of choice in a single command. The following command line options are available:

`-loadOntology` *`file`* Load and preprocess ontology from file

`-classifyOntology` Classify the ontology

`-outputSupers` *`file`* Output the sets of all super-classes

`-outputTaxonomy` *`file`* Output the direct sub- and super-classes

`-outputHierarchy` *`file`* Output the hierarchy as an indented tree

`-digServer` *`port`* Start CEL as a backend reasoner with a DIG server on the specified port, defaulted to 8080

`-quit` Exit; Don't enter the interactive interface

`-help` Display the help message

As an example, running CEL with the following command line options:

```
$ cel -loadOntology med.tbox -classifyOntology
      -outputTaxonomy med.dag -quit
```

will load and classify the ontology "med.tbox", and after the classification is done, output the subsumption DAG to "med.dag".

3 Macros and Functions

In this section, we give a detailed description of all commands offered by CEL. The commands are arranged in five groups.

3.1 Concept and Role Construction

In CEL, concept and role names can be used without prior declaration. To form concept and role names, we recommend the use of alpha-numerical characters, probably including symbols such as "_" and "-". Per default, CEL does not distinguish capital letters from lower-case ones. All symbols will automatically be capitalized when processing. In case of the need for case distinction, symbols shall be embraced within a pair of vertical bars, that is, `MALE` is equivalent to `Male`, but `|MALE|` is not to `|Male|`.

Complex concepts and roles can be constructed using the constructors of $\mathcal{EL}^+$ as summarized in the grammar on Page 3. In the following, the syntax is described in more detail.

---

top *keyword*

Description: The predefined, most general concept of any ontology, the top concept ($\top$, aka logical truth).
Syntax: `top`
OWL syntax: `owl:Thing`
Type: concept constructor

---

bottom *keyword*

Description: The predefined, most specific concept of any ontology, the bottom concept ($\bot$, aka logical falsity).
Syntax: `bottom`
OWL syntax: `owl:Nothing`
Type: concept constructor

and *keyword*

Description: N-ary conjunction ($C_1 \sqcap \cdots \sqcap C_n$)
Syntax: `(and` $C_1 \ldots C_n$`)`
OWL syntax: `owl:intersectionOf`
Example: `(and Human Male Drunk)`
Type: concept constructor

some *keyword*

Description: Existential restriction ($\exists RN.C$)
Syntax: `(some` RN C`)`
OWL syntax: `owl:someValuesFrom`
Example: `(some has-child Male)`
Remarks: Only atomic roles are allowed in an existential restriction
Type: concept constructor

compose *keyword*

Description: Binary role composition ($RN_1 \circ RN_2$)
Syntax: `(compose` RN_1 RN_2`)`
OWL syntax:
Example: `(compose has-parent has-brother)`
Type: role constructor

3.2 Knowledge Base Declarations

This section introduces the commands used to build up ontologies. There are two kinds of role axiom declarations and four kinds of concept axiom declarations.

define-primitive-role *macro*

Description: Defines a role name and its properties such as its super-roles and transitivity

Syntax:
```
(define-primitive-role RN &key (transitive nil)
                               (parent nil)
                               (parents nil)
                               (right-identity nil)
                               (left-identity nil))
                               (domain top))
```

OWL syntax: `owl:TransitiveProperty, rdfs:subPropertyOf, rdfs:domain`

There are no OWL equivalents to the expressivity right- and left-identity axioms.

Arguments: *RN* - role name
:transitive - if set to t declares that the role name is transitive.
:parent - specifies a super-role, or alternatively, use :parents with a singleton set, see examples.
:parents - specifies a list of super roles.
:right-identity - specifies a right-identity role for *RN*.
:left-identity - specifies a left-identity role for *RN*.
:domain - specifies the role domain restriction of *RN*.

Example:
```
(define-primitive-role has-descendant :transitive t
                                      :domain Human)
(define-primitive-role conj-role :parents (r1 ... rn))
(define-primitive-role cont-in :right-identity part-of)
```

Remarks: This macro integrates several DL role axioms. As an example, we would need two DL role axioms for the first example above: one for the transitivity ($hasDescendant \circ hasDescendant \sqsubseteq hasDescendant$) and one for the role inclusion ($hasDescendant \sqsubseteq hasChild$). The arguments *:left-identity* and *:right-identity* are not part of the KRSS standard.

role-inclusion *macro*

Description: Asserts an inclusion between a (possibly composite) role and a role name.
Syntax: `(role-inclusion` *R RN*`)`
OWL syntax: There is no OWL equivalent to this axiom with a role composition
Arguments: *R* - role name or binary role composite
RN - role name
Example:
```
(role-inclusion has-daughter has-child)
(role-inclusion (compose has-father has-sister)
has-aunt)
```
Remarks: This is a generalization of `define-primitive-role`. Role hierarchy, transitivity, and right-identity can be expressed in a more elegant way by using `define-primitive-role`. This macro is not part of the KRSS standard.

---

define-primitive-concept *macro*

Description: Asserts a subsumption relation between a concept name and a complex concept.
Syntax: `(define-primitive-concept` *CN C*`)`
OWL syntax: `rdfs:subClassOf`
Arguments: *CN* - concept name
C - (possibly complex) concept
Examples:
```
(define-primitive-concept Father Man)
(define-primitive-concept GrandMother (and Mother
Female))
```
Remarks: This macro states the necessary conditions for *CN*.

---

define-concept *macro*

Description: Defines a concept name.
Syntax: `(define-concept` *CN C*`)`
OWL syntax: `owl:equivalentClass`
Arguments: *CN* - concept name
C - (possibly complex) concept
Examples:
```
(define-concept Father (and Man (some has-child
Human)))
```
Remarks: This macro states both the necessary and sufficient conditions for *CN*. In CEL, a concept name may have multiple definitions.

concept-inclusion , implies *macro*

Description: Asserts a general concept inclusion (GCI) between two concepts.
Syntax: `(implies` C_1 C_2`)`
OWL syntax: `rdfs:subClassOf`
Arguments: C_1, C_2 - (possibly complex) concepts
Examples: `(implies (and Man (some has-sibling Parent)) Uncle)`
Remarks: The equivalent DL notation is $C_1 \sqsubseteq C_2$.

equivalent *macro*

Description: Asserts an equivalence between two concepts.
Syntax: `(equivalent` C_1 C_2`)`
OWL syntax: `owl:equivalentClass`
Arguments: C_1, C_2 - (possibly complex) concept
Examples: `(equivalent (and Man (some has-sibling (some has-child Female)) (and Uncle (some has-niece top))))`
Remarks: The equivalent DL notation is $C_1 \equiv C_2$.

disjoint *macro*

Description: Asserts disjointness of two or more concepts.
Syntax: `(disjoint` C_1 $\cdots$ C_2`)`
OWL syntax: `owl:disjointWith`
Arguments: C_i - (possibly complex) concept
Examples: `(disjoint Man Woman Male Female)`
Remarks: The equivalent DL notation is $C_1 \sqcap \cdots \sqcap C_n \sqsubseteq \bot$.

3.3 Operations on the Ontology (TBox)

clear-ontology , clear-tbox *macro*

Description: Removes all axioms and initializes everything.
Syntax: `(clear-tbox)`
Remarks: This macro is internally called whenever a new ontology is loaded.

load-ontology , load-tbox *macro*

Description: Reads in and and normalizes the input ontology.
Syntax: `(load-tbox` *tboxfile*`)`
Arguments: *tboxfile* - path of the input ontology file. If a filename is given without path, the ontology is assumed to be either in the current directory, or in the subdirectory `./tboxes`.
Remarks: The file specified by `tbox-file` is assumed to contain a list of commands as shown in Subsection 3.2. After having successfully loaded the ontology, it is ready to be classified, see the next command.

classify-ontology , classify-tbox *macro*

Description: Classifies the loaded and normalized ontology.
Syntax: `(classify-tbox)`
Remarks: Classify the default ontology in *polynomial* time.

ontology-prepared? , tbox-prepared? *macro*

Description: Checks if the ontology is prepared for classification.
Syntax: `(tbox-prepared?)`
Remarks: This is a fundamental requirement for `(classify-tbox)`.

ontology-classified? , tbox-classified? *macro*

Description: Checks if the ontology is successfully classified.
Syntax: `(tbox-classified?)`
Remarks: If this test is negative and concept subsumption is queried, the lazy subsumption approach is invoked.

ontology-consistent? , tbox-consistent? *macro*

Description: Checks whether or not the ontology is consistent, i.e. free from logical contradictions.
Syntax: `(tbox-consistent?)`
Remarks: This macro is available only after the ontology has been classified.

3.4 Query Commands

concept? *macro*

Description: Checks if the given name is known as a concept name.
Syntax: `(concept?` *CN*`)`
Arguments: *CN* - name to be tested

all-concepts *macro*

Description: Retrieves the list of all concept names occurring in the ontology.
Syntax: `(all-concepts)`

concept-subsumes? , subsumes? *macro*

Description: Queries if one concept name subsumes the other.
Syntax: `(concept-subsumes?` CN_1 CN_2`)`
Arguments: CN_1, CN_2 - concept names
Remarks: This macro only performs a lookup in the preclassified concept hierarchy; it cannot be called before the test `(tbox-classified?)` is positive. DL equivalent notation is $CN_2 \sqsubseteq^?_{\mathcal{T}} CN_1$.

In the current version of CEL, it is not possible to query the subsumption of *complex* concepts w.r.t. the classified ontology. To perform such a subsumption query, introduce two new names for the complex concepts in the subsumption query using *define-concept*, re-classify the augmented ontology, and then use *concept-subsumes?* to check subsumption between the newly introduced concept names.

concept-implies? , implies? *macro*

Description: Queries if one concept name implies (is subsumed by) the other.
Syntax: `(concept-implies?` CN_1 CN_2`)`
Arguments: CN_1, CN_2 - concept names
Remarks: Identical to *concept-subsumes*, but with swapped arguments

concept-satisfiable? , satisfiable? *macro*

Description: Queries if a concept is consistent, i.e. it can have an instance.
Syntax: `(concept-satisfiable?` CN`)`
Arguments: CN - concept name
Remarks: Alternatively, one could query `(concept-subsumes? bottom` CN`)` or `(concept-implies?` CN `bottom)`. If the anwer is `nil`, CN is satisfiable, and vice versa.

concept-equivalent? , equivalent? *macro*

Description: Queries if two concept names are equivalent (i.e., subsume each other)
Syntax: `(concept-equivalent?` CN_1 CN_2`)`
Arguments: CN_1, CN_2 - concept names

parents *macro*

Description: Returns a list of direct subsumers of a concept name.
Syntax: `(parents` *CN*`)`
Arguments: *CN* - concept name
Remarks: Requires DAG representation, which is computed on demand only when needed.

children *macro*

Description: Returns a list of direct subsumees of a concept name.
Syntax: `(children` *CN*`)`
Arguments: *CN* - concept name
Remarks: Requires DAG representation, which is computed on demand only when needed.

ancestors , super-concepts *macro*

Description: Returns a list of all (direct and indirect) subsumers of a concept name.
Syntax: `(ancestors` *CN*`)`
Arguments: *CN* - concept name

descendants , sub-concepts *macro*

Description: Returns a list of all (direct and indirect) subsumees of a concept name.
Syntax: `(descendants` *CN*`)`
Arguments: *CN* - concept name

role? *macro*

Description: Checks if the given name is known as a role name.
Syntax: `(concept?` *RN*`)`
Arguments: *RN* - name to be tested

all-roles *macro*

Description: Retrieves the list of all role names occurring in the ontology.
Syntax: `(all-roles)`

role-subsumes? *macro*

Description: Queries if one role name subsumes the other.
Syntax: `(role-subsumes?` RN_1 RN_2`)`
Arguments: RN_1, RN_2 - role names
Remarks: DL equivalent notation is $RN_2 \sqsubseteq^?_{\mathcal{T}} RN_1$.

role-implies? *macro*

Description: Queries if one role name implies (is subsumed by) the other.
Syntax: `(role-implies?` RN_1 RN_2`)`
Arguments: RN_1, RN_2 - role names
Remarks: Identical to *role-subsumes?*, but with swapped arguments

transitive? *macro*

Description: Checks if a role name is specified as transitive.
Syntax: `(transitive?` RN`)`
Arguments: RN - role name

super-roles *macro*

Description: Returns a list of all (direct and indirect) super roles.
Syntax: `(super-roles` RN`)`
Arguments: RN - role name

sub-roles *macro*

Description: Returns a list of all (direct and indirect) sub-roles.
Syntax: `(sub-roles` RN`)`
Arguments: RN - role name

3.5 Other Commands

start *function*

Description: Loads, preprocesses, and classifies the ontology, all in one command.
Syntax: `(start` *tboxfile* `&key (verbosity 2))`
Arguments: *tboxfile* - path string pointing to the input ontology file.
verbosity - the level of verbosity CEL talks to its interactive users. 0 or `nil` for none, 1 for crucial messages, 2 for normal messages, `t` for everything.
Remarks: See `load-ontology` and `classify-ontology`.

output-subsumption *macro*

Description: Outputs all computed subsumption relationships pair by pair.
Syntax: `(output-subsumption [`*filename*`])`
Arguments: *filename* - file name, possibly with path, to which the subsumption relationships are written. If not given, the output is written on the console screen.

output-imp-sets *macro*

Description: Outputs all computed subsumption relationships in the form of implication sets. In order to give succinct representation of the implication sets, this macro suppresses the tautology concepts, i.e. concepts subsuming the top concept, from being displayed in every set.
Syntax: `(output-imp-sets [`*filename*`])`
Arguments: *filename* - file name, possibly with path, to which the subsumption relationships are written. If not given, the output is written on the console screen.
Remarks: Obsolete and may be dispensed in the future version; compare `output-supers`.

output-supers *macro*

Description: Outputs all computed subsumption relationships in the form of subsumer sets. That is, for each concept A, the set of all A's subsumers is given.
Syntax: `(output-supers [`*filename*`])`
Arguments: *filename* - file name, possibly with path, to which the subsumption relationships are written. If not given, the output is written on the console screen.
Remarks: This is a complete representation of implication sets.

output-synonyms *macro*

Description: Outputs sets of told synonyms and their representative.
Syntax: `(output-synonyms [`*filename*`])`
Arguments: *filename* - file name, possibly with path, to which the subsumption relationships are written. If not given, the output is written on the console screen.

output-hierarchy *macro*

Description: Outputs a visualized representation of the computed taxonomy.
Syntax: `(output-hierarchy [`*filename*`])`
Arguments: *filename* - file name, possibly with path, to which the subsumption relationships are written. If not given, the output is written on the console screen.
Remarks: Requires DAG representation, which is computed on demand only when needed.

output-taxonomy *macro*

Description: Outputs the computed Hasse diagram by identifying the parents and children of each concept name.
Syntax: `(output-taxonomy [`*filename*`])`
Arguments: *filename* - file name, possibly with path, to which the subsumption relationships are written. If not given, the output is written on the console screen.
Remarks: Requires DAG representation, which is computed on demand only when needed.

4 An Example: Learning by Doing

In this section, we illustrate the use of CEL step by step. This includes

- creating an ontology,
- starting up the reasoner,
- loading the ontology,
- classifying the ontology,
- querying subsumptions, and
- output the classification.

Creating an Ontology CEL is a reasoning backend, and as such does not provide an ontology editor to create and maintain ontologies. Any ontology editor that produces KRSS syntax and supports (a subset of) the expressive means provided by $\mathcal{EL}^+$ can be used together with CEL. To get started, it is perhaps advisable to generate some simple ontologies using a standard text editor such as emacs.

The following simple ontology is provided as the file `./tboxes/med.tbox` of the CEL distribution. The concept name `HDSNT` is an abbreviation for "heart desease needing treatment".

```
(define-primitive-role cont-in :parent comp-of)

(implies Pericardium (and Tissue
                          (some cont-in Heart)))
(implies Pericarditis (and Inflammation
                           (some has-loc Pericardium)))
(implies Inflammation (and Disease
                           (some acts-on Tissue)))
(implies (and Disease
              (some has-loc
                    (some comp-of Heart)))
         (and Heartdisease
              (some has-state NeedsTreatment)))

(define-concept HDSNT
                (and Heartdisease
                     (some has-state NeedsTreatment)))
```

Starting up the reasoner CEL can be started by simply calling the executable in the `./bin` subdirectory of the distribution. After the reasoner has been successfully loaded, the greeting screen will appear and CEL will prompt for commands from the user.

Loading the ontology At the prompt, write down the following command:

```
(load-ontology "med.tbox")
```

CEL searches for the specified ontology not only in the current directory but also in the directory `./tboxes`. A full path to the ontology file can also be given.

Classifying the Ontology After having read in and preprocessed the ontology, CEL is ready to classify it. The user may want to check this status by asking

```
(ontology-prepared?)
```

The answer will then be `t`, i.e., "yes". At this point, the classification can be started by calling the command

```
(classify-ontology)
```

This may take some time depending on the size and complexity of the ontology. To load and classify an ontology in one step, the `start` function can be used.

Querying Subsumption When the ontology has successfully been classified (to check this, call `(ontology-classified?)`), subsumption between two concept names w.r.t. the classified ontology can be queried. In the example ontology above, the user might want to know if `Pericarditis` is in fact a heart disease needing treatment. This can be checked by querying

```
(concept-subsumes? HDSNT Pericarditis).
```

The set of all subsumers and subsumees of a concept name can also be obtained by using `ancestors` and `descendants`, respectively.

Output the Classification After the classification, the concept hierarchy can be archived in a file or displayed on screen by using the functions `output-subsumption`, `output-imp-sets`, and `output-supers`, depending on the format and brevity needed. If the classification has been carried out in mode 1, it is additionally possible with `output-taxonomy` to output the most compact representation of a taxonomy, i.e., Hasse diagram. By using `output-hierarchy`, a visualization of the terminological hierarchy can be displayed. Moreover, (maximal) sets of pairwise subsuming concept names (called *synonyms*) can be displayed by the function `output-synonyms`. Please refer to the more detailed explanation in Section 3.5.

References

[1] F. Baader, S. Brandt, and C. Lutz. Pushing the $\mathcal{EL}$ envelope. In *IJCAI-05*, Edinburgh, UK, 2005. Morgan-Kaufmann Publishers.

[2] Franz Baader, Carsten Lutz, and Boontawee Suntisrivaraporn. Is tractable reasoning in extensions of the description logic $\mathcal{EL}$ useful in practice? In *Proceedings of the 2005 International Workshop on Methods for Modalities (M4M-05)*, 2005.

[3] I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[4] D. L. McGuinness and F. van Harmelen. OWL web ontology language overview. See http://www.w3.org/TR/owl-features/, 2004.

[5] P. Patel-Schneider and B. Swartout. Description-logic knowledge representation system specification from the krss group of the arpa knowledge sharing effort. Technical report, DARPA Knowledge Representation System Specification (KRSS) Group of the Knowledge Sharing Initiative, 1993.